Descartes: a PITest engine to detect pseudo-tested methods - Tool Demonstration

Oscar Luis Vera-Pérez, Martin Monperrus, Benoit Baudry

Descartes: A PITest Engine to Detect Pseudo-Tested Methods

Tool Demonstration

Oscar Luis Vera-Pérez
Inria Rennes - Bretagne Atlantique
Rennes, France
oscar.vera-perez@inria.fr

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

ABSTRACT

Descartes is a tool that implements extreme mutation operators and aims at finding pseudo-tested methods in Java projects. It leverages the efficient transformation and runtime features of PITest. The demonstration compares Descartes with Gregor, the default mutation engine provided by PITest, in a set of real open source projects. It considers the execution time, number of mutants created and the relationship between the mutation scores produced by both engines. It provides some insights on the main features exposed by Descartes.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

pseudo-tested methods, extreme mutation, mutation testing, software testing, PITest

1 INTRODUCTION

Mutation analysis or mutation testing [2] evaluates the fault detection capabilities of a test suite. It does so by inserting artificial bugs in the form of subtle code changes. Then, it verifies if the test suite is able to detect those changes. The usual outcome from this analysis is the mutation score, that is, the ratio of planted faults (mutants) that has been detected to the total of mutants created.

Niedermayr and colleagues [5] recently introduced extreme mutation analysis. It is an alternative to traditional mutation that performs more coarse-grained transformations by eliminating, at once, all side effects of a method. For a *void* method this approach removes all instructions from its body. If the method is not *void*, then the body is replaced by a single *return* instruction with a predefined value. Listing 1 shows a simple Java method and Listing 2 shows two variants or mutants that could be created for this method using extreme mutation, in this case with constants 0 and 1. Besides removing all side effects, the technique ensures that the mutated method will always return the same value.

Extreme mutation addresses two challenges of the traditional approach. It creates much less mutants and can automatically avoid most transformations that could be equivalent to the original code. These two aspects are usually quoted as drawbacks that prevent the wide use of mutation testing in practice [3, 4]. Another benefit of this approach is that it operates at the method level which eases the understanding of the underlying testing problem. In addition to the mutation score, extreme mutation pinpoints a list of worst tested methods. In particular, the technique higlights methods executed by the test suite but where no extreme mutant is detected while running the tests. These methods are labeled as pseudo-tested in the work of Niedermayr et. al.[5].

In this demonstration, we present Descartes, an extreme mutation engine for PITest [1], a state-of-the-art mutation testing tool for Java projects. PITest is a popular tool that works with all major build systems: Ant, Gradle, Maven and can handle JUnit and TestNG test suites. Descartes brings a set of extreme mutation operators to PITest and discovers pseudo-tested methods. We also compare the result provided by Descartes with the outcome of Gregor, the default mutation engine for PITest. Our goal is to determine if extreme mutation can be used as a viable trade-off between code coverage, which assesses only test inputs, and traditional mutation analysis, which also addresses the oracles but at a very high cost. This is a novel contribution with respect to the work of Niedermayr et al whose focus is on checking whether code coverage is a good indicator of test quality when discerning between system and unit tests.

```
1  //Original method
   public static long factorial(int n) {
3      if(n==0) return 0;
       long result = 1;
5      for(int i = 2; i <= n; i++)
           result *= i;
7      return result;
   }
```

Listing 1: A simple Java method.

```
   //Extreme mutant 1
2  public static long factorial(int n) { return 0; }

4  //Extreme mutant 2
   public static long factorial(int n) { return 1; }
```

Listing 2: Two mutans created with extreme mutation.

2 AN OVERVIEW OF DESCARTES

Descartes is a tool to automatically detect pseudo-tested methods in Java programs tested with JUnit test suites. This detection relies on extreme mutation analysis. We implement this analysis as a mutation engine for PITest. In PITest's jargon, a mutation engine is a plugin that handles the discovery and creation of mutants. Such a plugin should also manage a set of mutation operators, which are models of the transformations to be performed.

Our extreme mutation engine provides a set of configurable mutation operators. A mutation operator is configured by specifying the literal value it should use to modify the method. Descartes supports literals of all Java primitive types, *String*, the *null* value and has two special operators: one to target *void* methods and another to return an empty array where possible. The engine does not mutate constructors.

Figure 1 illustrates the interaction between PITest and Descartes. PITest handles the inspection of the target project to discover all dependencies, creates execution units composed by the mutants and the tests to be executed, and ultimately runs the test cases. The mutation engine leverages all these functionalities and handles mutant discovery and creation.

Relying on the infrastructure and architecture of PITest allowed us to speed up the development of Descartes and its adoption in production. So far, the biggest challenges we have faced have been: 1) The lack of documentation describing how to create mutation engines for PITest; 2) The design and implementation of meaningful mutation operator abstractions and their interaction with the rest of the PITest framework; 3) Maintaining the engine up to date with the regular changes and releases of PITest; 4) Making the tool useful to developers. To overcome this last challenge we have augmented Descartes with custom reporting capabilities and functionalities to reduce the number of potential false positives, for example, we provide method filters based on the method structure rather than its signature.

To the best of our knowledge, Descartes is the only available alternative to the default engine provided by PITest. Our project could be used as an additional supporting material for those who are willing to create their own extensions.

3 DESCARTES VS GREGOR

Gregor is the default mutation engine for PITest. It provides most traditional mutation operators [1]. These operators work at the instruction level. Listing 3 shows examples of the transformations that can be produced by Gregor over the method exposed in Listing 1. The first variant of the method, shown in line 2 negates the condition in line 3. The second variant, shown in line 11, modifies the return value by adding 1 in line 16.

```
1       //Mutant 1. Changes == by !=
        public static long factorial(int n) {
3           if(n!=0) return 0;
            long result = 1;
5           for(int i = 2; i <= n; i++)
                result *= i;
7           return result;
        }
9
        //Gregor mutant 2. Changes the result value by adding 1
11      public static long factorial(int n) {
            if(n==0) return 0;
13          long result = 1;
            for(int i = 2; i <= n; i++)
15              result *= i;
            return result + 1;
17      }
```

Listing 3: Examples of mutants produced by Gregor.

We compare the execution of Gregor and Descartes in a selection of Java projects. These are all projects that use Maven as main build system, JUnit as main testing framework and are available form a version control hosting service, mostly Github.

Table 2 shows the metrics recorded for the comparison. For each mutation engine the table shows the execution time and number of mutants created. The "*Covered*" columns show the number of mutants actually executed by the test suite and planted in methods that were mutated by both engines. This distinction removes from the comparison mutants that Gregor may create in methods not analyzed by Descartes, and vice versa. For example, mutants created in constructors are left out. The "*Killed*" columns contain the number of mutants from the respective "*Covered*" column that were detected (killed) by the test suite. The "*Score*" columns show the corresponding mutation score, that is the ratio of "*Killed*" to "*Covered*".

[1]The full list is available here: http://pitest.org/quickstart/mutators/

Table 1: Extreme mutation operators used in the comparison.

Method type	Transformations
void	Empties the method
Reference types	Returns null
boolean	Returns true or false
byte,short,int,long	Returns 0 or 1
float,double	Returns 0.0 or 0.1
char	Returns ' ' or 'A'
String	Returns "" or "A"
T[]	Returns new T[]{}

For Gregor, all standard mutation operators were used. Descartes used the same mutation operators as Niedermayrs et. al. [5] plus two additional transformations, one to return *null* for reference types and another to return an empty array. The full list of extreme mutation operators is shown in table 1.

One can observe that Descartes creates much less mutants than Gregor which is reflected in the difference between the times to execute the analysis of each engine. In all cases, Descartes completed the task in much less time. Some interesting contrasts in this matter come from projects like `Spoon` where Descartes took a little less than two hours and a half while Gregor took more than 56 hours, `Java Git` with one hour and a half for extreme mutation and 16 hours for Gregor and `Jaxen XPath Engine` with less than two minutes against nearly 25 minutes. While the number of mutants created and covered affects the execution time, the tests themselves play an important role as they can involve heavy computation. Take, for example, the difference between `Apache Commons Lang` and `SCIFIO` with similar numbers of mutants and very different execution times.

As for the scores, one can notice that there is a certain correlation between the values obtained by both engines. Figure 2 shows a scatter plot, in which each point represents a project. The coordinates for each point are given by the scores, the x axis corresponds to the score from Descartes while the y represents the score from Gregor. The figure corroborates the tendency for a positive monotonic correlation between both scores, which means that, if the score with Gregor is high, it is more likely that the mutation score with Descartes will be also high. The Spearman correlation coefficient results in 0.6 for the projects studied with a *p-value* of 0.003, which indeed indicates that there is a moderate positive correlation. Anyways, there are cases such as `SCIFIO` and `XWiki Rendering Engine` which produce a medium to low mutation score with Gregor and scores above 83% with Descartes.

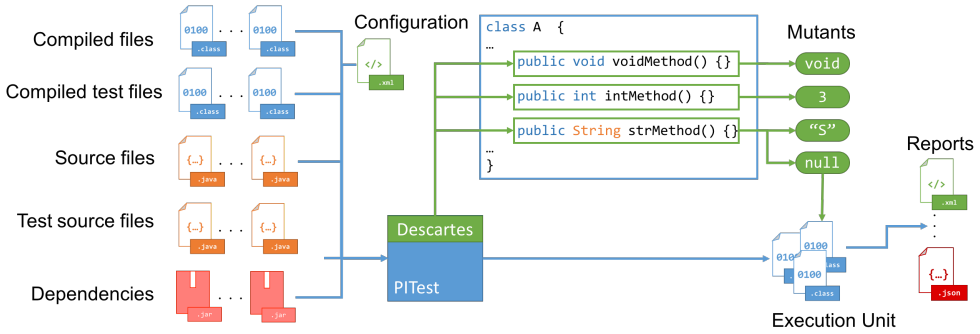

Figure 1: Interconnection between PITest and Descartes.

Table 2: List of projects used to compare both engines, the execution time for the analysis, the number of mutants created, mutants covered and placed in methods targeted by both tools, mutants killed and the mutation score.

	Descartes					Gregor				
Project	Time	Created	Covered	Killed	Score	Time	Created	Covered	Killed	Score
AuthZForce PDP Core	0:08:00	626	378	358	94.71	1:23:50	7296	3536	3188	90.16
Amazon Web Services SDK	1:32:23	161758	3090	2732	88.41	6:11:22	2141689	17406	13536	77.77
Apache Commons CLI	0:00:13	271	256	246	96.09	0:01:26	2560	2455	2183	88.92
Apache Commons Codec	0:02:02	979	912	875	95.94	0:07:57	9233	8687	7765	89.39
Apache Commons Collections	0:01:41	3558	1556	1463	94.02	0:05:41	20394	8144	7073	86.85
Apache Commons IO	0:02:16	1164	1035	968	93.53	0:12:48	8809	7633	6500	85.16
Apache Commons Lang	0:02:07	3872	3261	3135	96.14	0:21:02	30361	25431	22120	86.98
Apache Flink	0:14:04	4935	2781	2373	85.33	2:29:45	43619	21350	16647	77.97
Google Gson	0:01:08	848	657	617	93.91	0:05:34	7353	6179	5079	82.20
Jaxen XPath Engine	0:01:31	1252	953	921	96.64	0:24:40	12210	9002	6041	67.11
JFreeChart	0:05:48	7210	4686	3775	80.56	0:41:28	89592	47305	28080	59.36
Java Git	1:30:08	7152	5007	4507	90.01	16:02:03	78316	54441	40756	74.86
Joda-Time	0:03:39	4525	3996	3827	95.77	0:16:32	31233	26443	21911	82.86
JOpt Simple	0:00:37	412	397	379	95.47	0:01:36	2271	2136	2000	93.63
jsoup	0:02:43	1566	1248	1197	95.91	0:12:49	14054	11092	8771	79.08
SAT4J Core	0:53:09	2304	804	617	76.74	10:55:50	17163	7945	5489	69.09
Apache PdfBox	0:44:07	7559	3185	2548	80.00	6:20:25	79763	32753	20226	61.75
SCIFIO	0:24:14	3627	1235	1158	93.77	3:12:11	62768	19615	9496	48.41
Spoon	2:24:55	4713	3452	3171	91.86	56:47:57	43916	34694	27519	79.32
Urban Airship Client Library	0:07:25	3082	2362	2242	94.92	0:11:31	17345	11015	8956	81.31
XWiki Rendering Engine	0:10:56	5534	3099	2594	83.70	2:07:19	112605	50472	26292	52.09

4 PSEUDO-TESTED METHODS

The results of extreme mutation are not limited to produce a score for a given project. The proposal of Niedermayr et. al. [5] classifies methods according to the extreme mutant detection. A method is said to be **pseudo-tested** if it is covered by the test suite but no related extreme mutant is killed. These methods are the worst tested in the code base. Extreme mutation provides a framework to detect such methods more efficiently than traditional mutation testing.

Listing 4 shows a method belonging to one of the projects included in table 2. It was found to be pseudo-tested by Descartes. Only two extreme mutations are required to detect that the value of this method is not correctly verified by the test suite, if verified at all, while Gregor created 45 mutants. This is an example of the utility of extreme mutation.

```
1  public static boolean isValidXmlChar(int ch) {
       return (ch == 0x9)
3        || (ch == 0xA)
         || (ch == 0xD)
5        || (ch >= 0x20 && ch <= 0xD7FF)
         || (ch >= 0xE000 && ch <= 0xFFFD)
7        || (ch >= 0x10000 && ch <= 0x10FFFF);
   }
```

Listing 4: Real example of a pseudo-tested method.

Nevertheless, the result of Descartes is coarse-grained. Methods where extreme mutants are killed are not exempt from having

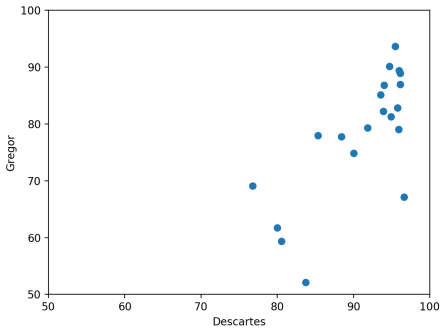

Figure 2: Visual correlation between scores

testing issues. Listing 5 shows a real example of a method where all extreme mutants were detected but Gregor created mutants that survived the analysis. In particular one of the traditional mutation operators changed the value of *Long.MIN_VALUE* in line 2. The modification was unnoticed by the test suite, which indicates that the corner case is not being tested. This level of detail can not be reached with the use of extreme mutation alone.

For a deep analysis regarding the utility in practice of Descartes in the search of pseudo-tested methods we invite the reader to check our work on the matter [6]. There, we analyze whether these methods are valid hints to improve existing test cases and we provide a set of testing issues found with the help of Descartes in real and well tested open-source projects.

```
    public long subtract(long instant, long value) {
2     if (value == Long.MIN_VALUE)
          throw new ArithmeticException(...);
4     return add(instant, -value);
    }
```

Listing 5: Example of a non pseudo-tested method.

5 DEMONSTRATION SCOPE

The demonstration will be directed to researchers and developers who wish to experiments with traditional and extreme mutation. It will be focused on the practical comparison of both mutation approaches. We will discuss how to interpret the results given by Descartes and how practitioners can use these results to enhance their test suites. We will show examples of real testing faults found with the use of the extreme mutation engine. The demo will also showcase the integration of Descartes and the latest check Github API [2] to discover pseudo-tested methods in commits and pull requests.

6 SUPPORTING MATERIALS

All materials related to the tool are available online. Here we provide a list with the main resources:

- **Descartes code repository**: Main code repository hosted in Github. It contains the code, documentation and instructions to build the tool.
  https://github.com/STAMP-project/pitest-descartes
- **Experimental data**: Consists in a set of files with the output obtained from both mutation engines as well as data concerning the studied projects.
  https://figshare.com/articles/data/6343280
- **Experimental material**: Github repository with additional experimental data and scripts to support the analysis and comparison of both mutation engines.
  https://github.com/STAMP-project/descartes-experiments
- **Maven Central artifacts**: Compiled versions of Descartes are available for use from Maven Central.
  https://mvnrepository.com/artifact/eu.stamp-project/descartes
- **Github Application repository**: Code of the prototype application to integrate Descartes in a Github repository.
  https://github.com/STAMP-project/descartes-github-app

[2] https://developer.github.com/v3/checks/

ACKNOWLEDGMENTS

This work has been supported by the EU Project STAMP ICT-16-10 No.731529.

REFERENCES

[1] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[2] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1979. Program mutation: A new approach to program testing. *Infotech State of the Art Report, Software Testing* 2, 1979 (1979), 107–126.

[3] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (Jan. 2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[4] Jakub Možucha and Bruno Rossi. 2016. Is Mutation Testing Ready to Be Adopted Industry-Wide?. In *Product-Focused Software Process Improvement (Lecture Notes in Computer Science)*. Springer, Cham, 217–232. https://doi.org/10.1007/978-3-319-49094-6_14

[5] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. 2016. Will my tests tell me if I break this code?. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. ACM Press, New York, NY, USA, 23–29. https://doi.org/10.1145/2896941.2896944

[6] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. 2018. A Comprehensive Study of Pseudo-tested Methods. *arXiv:1807.05030 [cs]* (July 2018). http://arxiv.org/abs/1807.05030 arXiv: 1807.05030.